Problem Statement:

Explain in brief

● What is the difference between memstore and hfile in HBase?

● Describe compactions in HBase.

● List and explain the logical entities in HBase.

● What will happen if we do not create a row key while inserting the data?

● How can filters be applied in HBase and what are the benefits?

● What are the data model operations in hBase?

● How can MapReduce be used with HBase?

● What is regionserver?

Q1. What is the difference between memstore and hfile in HBase?

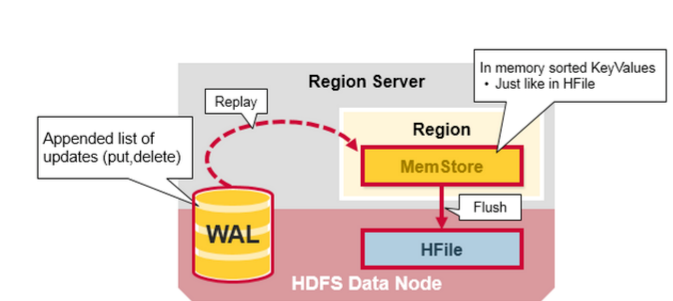
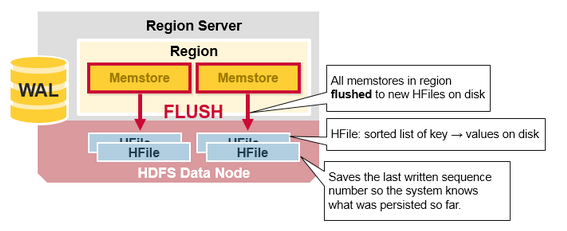
* A Hbase Store hosts a MemStore and 0 or more StoreFiles (HFiles). A Store corresponds to a column family for a table for a given region.
* The Write Ahead Log (WAL) records all changes to data in HBase, to file-based storage. If a RegionServer crashes or becomes unavailable before the MemStore is flushed, the WAL ensures that the changes to the data can be replayed.
* With a single WAL per RegionServer, the RegionServer must write to the WAL serially, because HDFS files must be sequential. This causes the WAL to be a performance bottleneck.
* WAL can be disabled to improve performance bottleneck. This is done by calling the Hbase client field

Mutation.writeToWAL(false)

***General Note***: Its general practice that while doing bulkloading data, WAL is disabled to get speed. But side effect is if you disable WAL you can’t get back data to replay if in case any memory crashes.

More over if you use solr+ HBASE + LILY, i.e. LILY Morphiline NRT indexes with hbase then it will work on WAL if you disable WAL for performance reasons, then Solr NRT indexing won’t work. Since Lily works on WAL.

Hbase architecture section

[](https://i.stack.imgur.com/F9qea.png)[](https://i.stack.imgur.com/snehl.png)

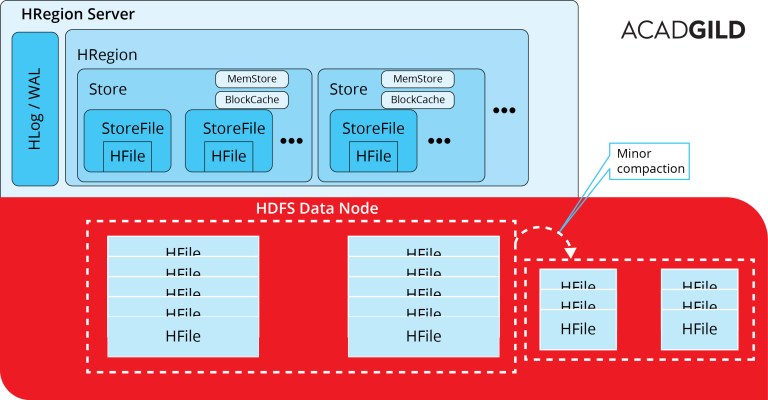
Q2. Describe compactions in HBase.

The servers are active all day long and while acting on Big Data, HBase hardly gets to write data. Therefore, it breaks the writing process into two parts: Minor Compaction and Major Compaction.

When the storage area of HBase is all most filled with data, it starts creating compressed files, which occupies less memory.

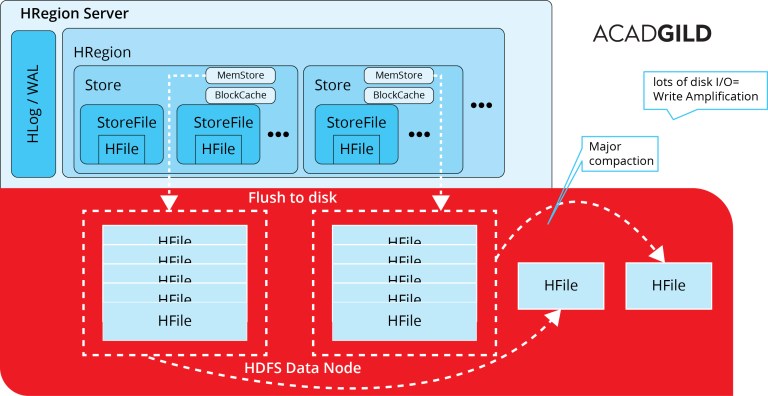
Here are the various processes involved in Minor Compaction:

* Bigger Hfile are created by combining smaller Hfiles.
* Hfile keeps the deleted file with them.
* Increases space in memory, useful to store more data.
* Merge sorting is used in process.



The other way to go around is major compaction

* Data present per column family in one region is accumulated to 1 Hfile.
* During this process, all deleted files or expired cells are deleted permanently
* Increase read performance of newly created Hfile.
* Accepts lots of I/O.
* Possibilities for traffic congestion.
* The Major compaction process is also known as Write Amplification Process.
* This process must be scheduled at a minimum bandwidth of network I/O.



**HBase compaction tuning tips**

**Short Description:**  
How to use some hidden HBase compaction configuration choices to enhance performance and stability of HBase cluster. Below

**Disabling automatic major compactions**

Usually, HBase users want to possess a full management of a major compaction events and the solely way to do that is to disable periodic automatic major compactions by setting hbase**.hregion.majorcompaction** to **0.**

But, sadly, this doesn’t provide you with 100% management of major compactions, because, sometimes, minor compactions can be promoted to major ones by HBase automatically, but, luckily, we’ve got another configuration choice, which will help during this case (below).

**Maximum compaction selection size**

We have another config option which can control compaction process:  
hbase.hstore.compaction.max.size (by default value is set to LONG.MAX\_VALUE)  
In HBase 1.2+ we have as well:  
hbase.hstore.compaction.max.size.offpeak

These choices control maximum size (in bytes) of compaction selection allowed. If you need to delay large compactions (major ones) until off-peak hours, you’ll set, for example:

hbase.hstore.compaction.max.size=500000000 (500MB)  
hbase.hstore.compaction.max.size.offpeak= 500000000000 (500GB)

The idea is to not enable minor compaction promotions to major ones throughout peak hours. Compactions can still happen during peak hours, however, they’ll be restricted to 500MB in size (or to whatever you set). Certainly, if your region size < 500MB some major compactions can still happen. We care about not the majority or minority of compaction here but about compaction size.

**Note:** when you run manual major compaction requests those settings are ignored

**Off peak compactions**

If your deployment has off peak hours you can use off-peak configuration settings.  
To enable off peak compaction following config options must be set :  
hbase.offpeak.start.hour= 0..23  
hbase.offpeak.end.hour= 0..23

Compaction file ratio for peak hours is 1.2, for off peak 5.0 (by default).

Both can be changed:

hbase.hstore.compaction.ratio

hbase.hstore.compaction.ratio.offpeak

Heigh the file ratio value – the more aggressive (frequent) compaction is going to be. Default values are fine for the majority of deployments.

Q3. What will happen if we do not create a row key while inserting the data?

Every row in an HBase table has a unique identifier called its rowkey (Which is equivalent to Primary key in RDBMS, which would be distinct throughout the table). Every interaction you are going to do in database will start with the RowKey only. If you do not specify row key then it will throw an error.

Q5.How can filters be applied in HBase and what are the benefits?

**HBase Filtering**

When reading data from HBase using Get or Scan operations, you can use custom filters to return a subset of results to the client. While this does not reduce server-side IO, it does reduce network bandwidth and reduces the amount of data the client needs to process. Filters are generally used using the Java API, but can be used from HBase Shell for testing and debugging purposes.

Q6.What are the data model operations in hBase?

The four primary data model operations are Get, Put, Scan, and Delete. Operations are applied via HTable instances.

**5.7.1. Get**

Get returns attributes for a specified row. Gets are executed via HTable.get.

**5.7.2. Put**

Put either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via HTable.put (writeBuffer) or HTable.batch (non-writeBuffer).

**5.7.3. Scans**

Scan allow iteration over multiple rows for specified attributes.

The following is an example of a on an HTable table instance. Assume that a table is populated with rows with keys "row1", "row2", "row3", and then another set of rows with the keys "abc1", "abc2", and "abc3". The following example shows how startRow and stopRow can be applied to a Scan instance to return the rows beginning with "row".

Q7.How can MapReduce be used with HBase?

 Using HBase as a data source and a sink (the destination for the output):

1. HBase provides a TableInputFormat, to which you provided a table scan, which splits the rows resulting from the table scan into the regions in which those rows reside.
2. The map process is passed an ImmutableBytesWritable that contains the row key for a row and a Result that contains the columns for that row.
3. The map process outputs its key/value pair based on its business logic in whatever form makes sense to your application.
4. The reduce process builds its results but emits the row key as an ImmutableBytesWritableand a Put command to store the results back to HBase.
5. Finally, the results are stored in HBase by the HBase MapReduce infrastructure. (You do not need to execute the Put commands.)

Q8. What is regionserver?

RegionServers are the software processes (often called daemons) you activate to store and retrieve data in HBase (Hadoop Database). In production environments, each RegionServer is deployed on its own dedicated compute node. When you start using HBase, you create a table and then begin storing and retrieving your data.

However, at some point — and perhaps quite quickly in big data use cases — the table grows beyond a configurable limit. At this point, the HBase system automatically splits the table and distributes the load to another RegionServer.

In this process, often referred to as *auto-sharding*, HBase automatically scales as you add data to the system — a huge benefit compared to most database management systems, which require manual intervention to scale the overall system beyond a single server. With HBase, as long as you have in the rack another spare server that’s configured, scaling is automatic!

Why set a limit on tables and then split them? After all, HDFS is the underlying storage mechanism, so all available disks in the HDFS cluster are available for storing your tables. (Not counting the replication factor, of course.) If you have an entire cluster at your disposal, why limit yourself to one RegionServer to manage your tables?

Simple. You may have any number of tables large or small and you’ll want HBase to leverage all available RegionServers when managing your data. You want to take full advantage of the cluster’s compute performance. Furthermore, with many clients accessing your HBase system, you’ll want to use many RegionServers to meet the demand.

HBase addresses all of these concerns for you and scales automatically in terms of storage capacity and compute power.

